Amendments to the Specification:

Please replace the paragraph beginning at page 2, line 25 with the following amended paragraph:

The pseudo-random verification framework may be run continuously to test emulation of the complete set of instructions from the native ISA. Even the least-used machine instructions are tested by the framework. Further, the framework automates the emulation verification process. Inter-machine communications allow the native and the target computer architectures to process the same machine instructions from the same initial states. Any inconsistencies in the final produced states indicate the emulation error. The framework can then easily pinpoint the exact machine instruction, register number and input machine state that caused the emulation error, thereby significantly reducing the amount of time required for debugging. Finally, the emulation errors detected using this framework are easily reproducible.

Please replace the paragraph beginning at page 4, line 25 with the following amended paragraph:

The framework 10 relies on pseudo-random generation of machine level instructions to comprehensively test the correct emulation of native applications on a second computer architecture. The machine level instructions may be generated according to many different pseudo-random generation schemes. In an embodiment, each machine level instruction is assigned a probability. In particular, a specific instruction to be generated is controlled by an input probability file, which is a list of pre-defined machine instructions, each with an associated probability. The machine instructions are arranged in a hierarchy form and divided into segments based on a function of the instruction. A first such segmentation may include CPU instructions, which include floating instructions ~~and CPU instructions~~. Continuing, the CPU instructions may be further segmented according to arithmetic/logic, immediate, memory, system, memory management, and branch instructions, and other CPU instructions. The arithmetic/logic instructions may be segmented into ADD, SUB, AND, OR, and XOR instructions, and other arithmetic/logic instructions, for example. Each hierarchical instruction is assigned a probability such that a cumulative probability along any hierarchical path equals 1.00 ~~1.0~~.

Please added the following paragraph at page 6, line 18:

Beginning at any node, the sum of probabilities of the immediate exiting branches (or paths) is 1.0. For example, beginning at the CPU node, the nonzero exiting branches are 0.25 + 0.25 + 0.5 = 1.0. Further, the probability of any specific machine instruction is the product of the probabilities of taking each branch that leads to said specific machine instruction. Note that the sum of the probabilities of all specific machine instructions should equal 1.0.

Please replace the paragraphs beginning at page 6, line 22 with the following amended paragraphs:

By assigning higher probability values to some instructions and lower probability values to other instructions, the test designer can ensure that all binary instructions are eventually tested after the testing continues for an infinite length of time.

The RCG 20 begins with a seed value and then determines a pseudo random probability value. ~~For example, a seed value of 10 may produce a probability sequence of 0.34, 0.8, ... Using the cumulative probability in the above example, a random probability of 0.34 would lead the RCG 20 to generate a cpu_immediate instruction, and a random probability of 0.8 would lead the RCG 20 to generate a pr_cpu_immediate_subi instruction.~~

Please replace paragraph beginning at page 7, line 9 with the following amended paragraph:

The main process begins at 100. In code generate block 110, the RCG 20 generates pseudo-random native machine code and generates a pseudo-random initial machine state. The native machine code and the initial machine state are provided to the native platform 11, and the native platform 11 is initialized using the initial machine state, block 120. The same native machine code and initial machine state are also provided to the target platform 12, and the target platform 12 is initialized, block 130. The native machine code and initial machine state may be provided to the target platform 12 using TCP/IP protocols, for example.

Please replace the abstract with the following amended paragraph:

A method and an apparatus allows complete and efficient verification of cross-architecture ISA emulation. A random verification framework runs concurrently on two different computer architectures. The framework operates without regard to existing native applications and relies instead on binary instructions in a native ISA. The framework determines emulation errors at a machine instruction level. A random code generator

generates one or more sequences of native machine instructions and corresponding initial machine states in a pseudo-random fashion. The native instructions are generated from an entire set of the native ISA. The instructions and the state information are provided to initialize a native computer architecture. The same instructions and state information are provided using standard machine-to-machine languages, such as TCP/IP, for example, to a target computer architecture. A binary emulator then translates the native instructions so that the instructions may be executed on the target computer. Alternatively, the binary emulator may be embodied as a software routine operating on a simulator, which in turn operates on the native computer architecture. The final states of the native and the target computer architectures are gathered, and a verification engine compares the results. Any differences may indicate an emulation error or failure. The random verification framework may be run continuously to test emulation of the complete set of instructions from the native ISA after the testing continues for an infinite length of time.